Query Processing: A Systems View

Introduction to Databases
CompSci 316 Fall 2019
Announcements (Wed., Nov. 13)

• Project milestone 2 feedback on Gradescope by Fri.
  • Weekly update due on Piazza today!
• Homework 4 due on before Thanksgiving Break
A query’s trip through the DBMS

SQL query

Parser

Parse tree

Validator

Logical plan

Optimizer

Physical plan

Executor

Result

SELECT name, uid
FROM Member, Group
WHERE Member.gid = Group.gid;

π_{name, uid} \ θ_{Member.gid=Group.gid}

Member Group

Member gid

Group gid

SCAN (Group)

SORT (gid)

MERGE-JOIN (gid)

PROJECT (name, gid)

... table...

Member Group... table...

Member Group... table...

SELECT list

<where-cond>

<from-list>

SFW

<Query>

Member gid

Group gid

...
Parsing and validation

• Parser: SQL → parse tree
  • Detect and reject syntax errors

• Validator: parse tree → logical plan
  • Detect and reject semantic errors
    • Nonexistent tables/views/columns?
    • Insufficient access privileges?
    • Type mismatches?
      • Examples: AVG(name), name + pop, User UNION Member

• Also
  • Expand *
  • Expand view definitions

• Information required for semantic checking is found in system catalog (which contains all schema information)
Logical plan

• Nodes are logical operators (often relational algebra operators)
• There are many equivalent logical plans

An equivalent plan:

π\(\text{Group.name}\)

σ\(\text{User.name} = \text{“Bart”} \land \text{User.uid} = \text{Member.uid} \land \text{Member.gid} = \text{Group.gid}\)

π\(\text{Group.name}\)

△\(\text{Member.gid} = \text{Group.gid}\)

△\(\text{User.uid} = \text{Member.uid}\)

σ\(\text{name} = \text{“Bart”}\)

△\(\text{Member}\)

△\(\text{Group}\)

△\(\text{User}\)
Physical (execution) plan

• A complex query may involve multiple tables and various query processing algorithms
  • E.g., table scan, index nested-loop join, sort-merge join, hash-based duplicate elimination…

• A **physical plan** for a query tells the DBMS query processor how to execute the query
  • A tree of **physical plan operators**
  • Each operator implements a query processing algorithm
  • Each operator accepts a number of input tables/streams and produces a single output table/stream
Examples of physical plans

SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;

• Many physical plans for a single query
  • Equivalent results, but different costs and assumptions!
    DBMS query optimizer picks the “best” possible physical plan
Physical plan execution

• How are intermediate results passed from child operators to parent operators?
  • **Temporary files**
    • Compute the tree bottom-up
    • Children write intermediate results to temporary files
    • Parents read temporary files
  • **Iterators**
    • Do not materialize intermediate results
    • Children pipeline their results to parents
Iterator interface

• Every physical operator maintains its own execution state and implements the following methods:
  • `open()`: Initialize state and get ready for processing
  • `getNext()`: Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  • `close()`: Clean up
An iterator for table scan

• State: a block of memory for buffering input $R$; a pointer to a tuple within the block

• `open()`: allocate a block of memory

• `getNext()`
  • If no block of $R$ has been read yet, read the first block from the disk and return the first tuple in the block
    • Or null if $R$ is empty
  • If there is no more tuple left in the current block, read the next block of $R$ from the disk and return the first tuple in the block
    • Or null if there are no more blocks in $R$
  • Otherwise, return the next tuple in the memory block

• `close()`: deallocate the block of memory
An iterator for nested-loop join

**R**: An iterator for the left subtree
**S**: An iterator for the right subtree

- **open()**
  - R.open()
  - S.open()
  - \( r = \text{R}.\text{getNext}() \)

- **getNext()**
  - while True:
    - \( s = \text{S}.\text{getNext}() \)
    - if s is null: # no more tuple from S
      - \( \text{S}.\text{close()} \) # reopen S
      - \( \text{S}.\text{open()} \)
      - \( s = \text{S}.\text{getNext}() \)
      - if s is null: # S is empty!
        - return null
      - \( r = \text{R}.\text{getNext}() \) # move on to next r
      - if r is null: # no more tuple from R
        - return null
      - if joins(r, s):
        - return concat(r, s)

- **close()**
  - R.close()
  - S.close()
An iterator for 2-pass merge sort

• **open()**
  • Allocate a number of memory blocks for sorting
  • Call open() on child iterator

• **getNext()**
  • If called for the first time
    • Call getNext() on child to fill all blocks, sort the tuples, and output a run
    • Repeat until getNext() on child returns null
  • Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
    • Return the smallest tuple and advance the corresponding pointer; if a block is exhausted bring in the next block in the same run

• **close()**
  • Call close() on child
  • Deallocate sorting memory and delete temporary runs
Blocking vs. non-blocking iterators

• A **blocking** iterator must call `getNext()` exhaustively (or nearly exhaustively) on its children before returning its first output tuple
  - Examples: sort, aggregation

• A **non-blocking** iterator expects to make only a few `getNext()` calls on its children before returning its first (or next) output tuple
  - Examples: dup-preserving projection, filter, merge join with sorted inputs
Execution of an iterator tree

• Call `root.open()`
• Call `root.getNext()` repeatedly until it returns null
• Call `root.close()`

Ϝ Requests go down the tree
Ϝ Intermediate result tuples go up the tree
Ϝ No intermediate files are needed
  • But maybe useful if an iterator is opened many times
    • Example: complex inner iterator tree in a nested-loop join; “cache” its result in an intermediate file
Iterators are showing their age...

While iterators are an elegant way of pipelining execution, their implementation tends to be inefficient on modern architectures

• Too many (virtual) function calls
• Poor data locality—in memory instead of CPU registers
• Fail to take advantage of
  • Compiler loop unrolling
  • CPU pipelining
  • SIMD (single instruction, multiple data)
Which one do you think runs faster?

class NLJ
    open()
    R.open()
    S.open()
    r = R.getNext()
    getNext()
        while True:
            s = S.getNext()
            if s is null:  # no more tuple from S
                S.close()  # reopen S
                S.open()
                s = S.getNext()
            if s is null:  # S is empty!
                return null
            r = R.getNext()  # move on to next r
            if r is null:  # no more tuple from R
                return null
            if joins(r, s):
                return concat(r, s)
    close()
    R.close()
    S.close()

class Aggr
    open()
    R.open()
    state = init()
    getNext()
        while True:
            r = R.getNext()
            if r is null:  # no more tuple from R
                return finalize(state)
            state = accumulate(state, r)
    close()
    R.close()

versus

count = 0
for r in R:
    for s in S:
        if r.A = s.A:
            count += 1
return count
Whole-stage “codegen”

• Given a physical plan, fuse operators together to generate query-specific code, with loops instead of iterator function calls

• Instead of “interpreting” the physical plan, give generated code to an optimizing compiler

☞ Functionality of a general-purpose execution engine; performance as if system is hand-built to run your specific query

• This approach has been adopted by newer systems, such as Spark